NRL Report 8661

# A Two-Priority Frame-Synchronization Algorithm

J. O. COLEMAN

*Radar Analysis Branch*
*Radar Division*

February 28, 1983

NAVAL RESEARCH LABORATORY
Washington, D.C.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

| | | |
|---|---|---|
| **REPORT DOCUMENTATION PAGE** | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
| 1. REPORT NUMBER<br>NRL Report 8661 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (*and Subtitle*)<br>A TWO-PRIORITY FRAME-SYNCHRONIZATION ALGORITHM | | 5. TYPE OF REPORT & PERIOD COVERED<br>Interim report on a continuing NRL problem |
| | | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(*s*)<br>J.O. Coleman | | 8. CONTRACT OR GRANT NUMBER(*s*) |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>Naval Research Laboratory<br>Washington, DC 20375 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>62712N; SF12131691;<br>53-0620-00 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Naval Sea Systems Command<br>Washington, DC 20362 | | 12. REPORT DATE<br>February 28, 1983 |
| | | 13. NUMBER OF PAGES<br>29 |
| 14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*) | | 15. SECURITY CLASS. (*of this report*)<br>UNCLASSIFIED |
| | | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (*of this Report*)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*)

Frame synchronization
Data link control protocols
Data link control procedures
Radar communication

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

Most digital communication involves the transmission of data in distinct and identifiable groups of bits, sometimes referred to as frames. In many types of systems, separate control channels for marking frame boundaries are unavailable and frame-synchronization information must be embedded in the data stream. The standard techniques for synchronization are inadequate in situations in which both of the following are true: (1) data can be transmitted only occasionally and in amounts not necessarily corresponding to frames; (2) data frames are divided into two categories, high priority

(Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

20. ABSTRACT *(Continued)*

and low priority, with high priority indicating time-critical information. In these situations the transmitter data queue should operate as a priority queue; i.e., it should operate in a first-in-first-out fashion except that when high-priority bits enter the queue they immediately move past any low-priority bits already in the queue. Because this can result in a high-priority frame being inserted into the middle of a partially transmitted low-priority frame, the synchronization algorithm in this context needs to do two things in a compatible way: (1) delimit low-priority frames and (2) mark high-priority frames in such a way that they can be detected anywhere in the low-priority data stream. This report describes such an algorithm.

UNCLASSIFIED

## CONTENTS

# A TWO-PRIORITY FRAME-SYNCHRONIZATION ALGORITHM

## INTRODUCTION

### The Problem

Most types of digital communication involve the transmission of data in distinct and identifiable *groups* of bits, variously referred to as *messages, packets,* or *frames.* When the physical channel over which the data are transmitted consists of a set of wires, the beginning and end of a frame can be indicated by the raising and lowering of a control signal on a wire dedicated to that task. With some types of channels, e.g., radio channels, it is inconvenient to provide a separate control channel, and frame demarcation must somehow be embedded in the data stream itself. When the transmitted data stream consists of characters (each character represented by a fixed bit pattern), special demarcation characters are often used to delimit blocks of data [1]. The data stream in many contexts, however, consists of blocks of bits that have no character structure. The data-link control procedures that have evolved in this context insert a unique bit pattern, called a marker or *flag sequence,* into the data to separate frames [2]. To achieve data transparency, it is then necessary to take special measures to prevent any coincidental occurrence of the flag sequence in the data itself from being interpreted as the end of the frame. These measures are referred to as *bit stuffing* or simply *stuffing.*

Carlson [3] describes the stuffing procedure used in the Advanced Data Communication Control Procedures [4] to achieve data transparency as follows:

> The flag sequence is a unique 8-bit pattern (a 0-bit followed by six 1-bits ending with a 0-bit) used to synchronize the receiver with the incoming frame...
>
> To achieve transparency, the unique flag sequence is prohibited from occurring anywhere in the...information...by having the transmitter and receiver perform the following action...
>
> Transmitter: insert a zero bit following five contiguous one bits anywhere before sending the closing flag sequence ('bit stuffing').
>
> Receiver: delete the zero bit following five contiguous one bits following a zero bit anywhere before receiving the closing flag sequence ('destuffing').

Similar procedures are widely used.

I recently encountered a system in which this type of scheme was inadequate. In an NRL program that is investigating the use of surveillance radars for jam-resistant communication [5], I discovered a need for a multiplexing/synchronization procedure that could combine data frames from two sources into a single, continuous data stream at the transmitter, and, of course, perform the inverse operation at the receiver. The "obvious" solution of using a standard framing and stuffing procedure with the source given at the beginning of each frame was not applicable. The reason is related to the delay inherent in the mechanical rotation of the radar antenna, as will become apparent in the following discussion.

Manuscript approved October 22, 1982.

The primary benefit gained by the use of a radar for communication is the jam resistance obtained by transmission through its high-power transmitter and its high-gain antenna. The associated penalty is the necessity of waiting for the slowly-rotating antenna to reach the desired position before transmitting. This implies long communication delays, making a priority structure for the data desirable. For example, data frames could be divided into two categories: high priority and low priority. The bulk of traffic would presumably be low priority, with the high-priority designation reserved for time-critical information or, possibly, link-control information.

After the antenna has reached the desired position for the beginning of a transmission, transmission must be limited in duration to the interval in which the antenna's beam is over the receiving site. Once that interval has passed, no data can be transmitted to that particular receiving site until the radar antenna has nearly completed another rotation. If as much of the data as possible is transmitted on a given pass of the antenna over the receiver, transmission will frequently have to cease somewhere in the middle of a data frame, rather than on a frame boundary. Therefore, some part of a frame is frequently *queued* at the transmitter, awaiting the arrival of the antenna at a particular position. What happens if a new frame arrives from the data source?

If the arriving frame is of the same priority as the one whose final portions are already queued for transmission, there is no problem. The remainder of the first frame can be transmitted, and then the transmission of the new frame can begin. The same is true if the new frame is of a lower priority than the first frame. If, however, a high-priority frame arrives when a preceding low-priority frame has been only partially transmitted, it is desirable that the high-priority frame be transmitted *before* the transmission of the low-priority frame is completed. In effect, this requires that the transmitter queue operate as a *priority queue;* i.e., it should operate in a first-in-first-out fashion *except* that when high-priority bits enter the queue they immediately move past any low-priority bits already in the queue. This tends to minimize the total delay experienced by high-priority frames relative to low-priority frames, with the delay difference sometimes corresponding to one or more multiples of the rotational period of the radar antenna.

Now consider the situation at the output of the receiver. Because the insertion of a high-priority frame into the priority queue puts it in front of any existing low-priority data, and because the low-priority data may be only part of a frame (if part of the frame has already been transmitted), a high-priority frame can appear inserted into the low-priority data stream at an unpredictable place. On the other hand, the reverse is not true. Presumably, frames will always be made available from the data source as complete units. Therefore, a high-priority frame can always be inserted into the priority queue as a unit. Since the priority queue ensures that the transmitter will always move all available high-priority data before moving any low-priority data, a high-priority frame will never have other data appear in its midst.

The picture the receiver sees is that of a continuous* stream of low-priority data, presumably divided into frames by a marking/stuffing procedure, with occasional high-priority frames inserted into the data stream at completely random locations. A high-priority frame could even end up in the middle of a flag sequence being used to delimit low-priority frames! To be able to correctly identify frame boundaries in spite of these difficulties an algorithm is required to do two things:

(1) delimit low-priority frames, and

(2) mark high-priority frames in such a way that they can be detected anywhere in the low-priority data stream.

* The physical breaking of the data into bursts corresponding to antenna rotations is irrelevant.

This situation is not unique to radar communication; it can occur anytime prioritized units of data (be they frames, messages, packets, or whatever) must be put through a system where queuing delays are longer than transmission times. This occurs, for example, with some "bursty" channels, especially if the prioritized units of data can be larger than the bursts or if the size of the bursts is unpredictable. Communication systems, such as meteor-scatter systems, that depend on intermittently available propagation modes sometimes fall into this category.

### The Approach to a Solution

The ADCCP marking and stuffing scheme described earlier, while adequate for a single data stream, is insufficient here because it does not provide a mechanism for handling the two data streams, low and high priority, that will be intermixed at the output of the priority queue and hence at the output of the receiver. A stuffing scheme similar to that just described *can* be used, however, as the basic synchronization mechanism for the low-priority data stream, with a flag sequence to signal the end of each low-priority frame. A way is then needed to detect the beginning of a high-priority frame. To do this, the stuffing algorithm can mark the beginning of each high-priority frame with a special "high-priority flag sequence." The end of the frame can be marked with the same end-of-frame flag sequence used for the low-priority data. Since the low-priority data (*including* the end-of-frame flag sequence) must also be protected from accidental occurrences of this high-priority flag sequence, it is necessary to perform protective stuffing operations on the low-priority data with respect to *both* flag sequences. In addition, it is desirable that the stuffing operations on the low- and high-priority data be identical if the flag sequence at the end of a high-priority frame is ever missed at the receiver (possible only with errors in transmission). In that case the flag sequence at the end of the surrounding low-priority frame would be recognized and would allow processing to revert to low-priority mode.

A marking-and-stuffing algorithm based on these ideas is presented below. An example is then presented to help clarify the operation of the algorithm. The body of the report concludes with an outline of the procedure used to prove that the algorithm derived will operate as desired for any input. The report includes two appendixes: Appendix A details the derivation of the algorithm, and Appendix B describes a test implementation used as a final check on the algorithm.

## THE TWO-PRIORITY FRAME-SYNCHRONIZATION ALGORITHM

A concise description of the algorithm is presented here. Separate descriptions are given of the transmitter and receiver portions of the algorithm. Readers wishing to understand how this algorithm was derived are referred to Appendix A.

### At the Transmitter

A priority queue is maintained for the data at the input to the transmitter. This is the equivalent of having two queues, a high-priority queue and a low-priority queue. Both of these equivalent queues operate strictly in a first-in-first-out fashion. When the transmitter is ready to send some data, it first takes data from the high-priority queue and only takes data from the low-priority queue if the high-priority queue becomes empty. If more data should arrive in the high-priority queue while transmission of low-priority data is in progress, the transmitter immediately reverts to taking its input from the high-priority queue until it is again emptied.

Frames arriving from the data source are first marked and stuffed according to their priority, high or low, as described below. High-priority frames are then placed in the high-priority queue, and low-priority frames are placed in the low-priority queue.

The procedure to mark and stuff a newly-obtained frame is as follows: Append a comma to the end of the frame and pass the frame to the finite-state-machine (FSM) stuffer of Table 1. The table is

Table 1 — The Stuffer

| State | Input='0' | Input='1' | Input=',' |
|---|---|---|---|
| AA | AA/0 | BB/1 | AA/011111010 |
| BB | AA/0 | CC/1 | AA/011111010 |
| CC | AA/0 | DD/1 | AA/011111010 |
| DD | AA/0 | EE/1 | AA/011111010 |
| EE | AA/0 | AA/100 | AA/011111010 |

interpreted as follows: The states of the machine are listed on the left, and the possible inputs are listed across the top. For each combination of state and input the table shows the machine's next state. When output is called for, it is shown following the next state and separated from it by a slash. The first state in the table is the starting state of the machine.*

This FSM changes each input comma to a flag sequence of 011111010. To prevent coincidental occurrences of this sequence in the data from being interpreted as the flag sequence, this machine stuffs two extra zeros into the data stream following each occurrence in the data of five consecutive ones.

Each state in Table 1 represents a situation in which the machine has encountered a particular input pattern but has not finished acting on that particular input pattern. For example, the stuffer of Table 1 will be in state CC when it has encountered two consecutive ones in the input but cannot yet decide whether they are part of a sequence of ones long enough to require stuffing zeros into the output. If three more consecutive ones follow, two extra zeros will be stuffed (from state EE). If, however, a zero arrives before three more ones have arrived, no stuff bits will be inserted. Thus, its being in state CC is equivalent to the FSM's *remembering* that it has seen two ones.

If (and only if) the frame is a high-priority frame, the output of the stuffer of Table 1 should be preceded by the binary (high-priority flag) sequence 1111110. The accumulated output of the stuffer should be inserted into the queue that corresponds to the priority of the frame being stuffed.

## At the Receiver

The data from the receiver should be passed to the two-priority destuffing machine of Table 2. The format of Table 2 is similar to that of Table 1, except that an additional column labeled **Saved** has been added. **Saved** is an auxiliary variable and will always contain the name of a state; it is initialized to AA, the starting state (only once, *not* once per frame). For (state) transitions on which input to the machine is equal to one, Table 2 is interpreted exactly like Table 1. For transitions on which the input to the machine is zero, the machine first determines the next state (without actually changing state) and generates the required output, if any. It then sets the variable **Saved** to the value shown in the **Saved** column of Table 2 and changes state. If there is no entry in the **Saved** column, the value of **Saved** is left unchanged. Because the input to the destuffer is provided by the output of the stuffer, and because the stuffer output consists of only zeros and ones, there can be no commas at the input of the destuffer. An output of "err" is shown with transitions that cannot take place without an error in transmission between the stuffer and the destuffer. In such cases the next state shown was chosen arbitrarily (this is discussed further in Appendix A).

There are two output buffers, **low-priority-buffer** and **high-priority-buffer,** and two auxiliary flags, **low-priority-error** and **high-priority-error.** Output is initially put into **low-priority-buffer.** Both auxiliary flags are initially clear. The exclamation-point output shown with certain transitions of Table 2 should be interpreted as: begin putting data into the **high-priority-buffer** instead of the **low-priority-buffer.** The "err" output shown with certain transitions should be interpreted as: set the appropriate

* This is a conventional way to represent a finite-state machine, for example, among digital-design engineers [6].

Table 2 — The Two-Priority Destuffer

| State | Input='0' | **Saved** | Input='1' |
|---|---|---|---|
| AA | AB | | BA |
| AB | AB/0 | | BB |
| BB | AB/01 | | CB |
| CB | AB/011 | | DB |
| DB | AB/0111 | | EB |
| EB | AB/01111 | | FB |
| FB | AG | | GB |
| AG | AA/011111 | | BG |
| BG | **Saved**/, | AA | CG |
| BA | AB/1 | | CA |
| CA | AB/11 | | DA |
| DA | AB/111 | | EA |
| EA | AB/1111 | | FA |
| FA | AM/11111 | | GA |
| AM | AA | | BM |
| GB | AA/! | AB | HB |
| HB | AA/! | BB | IB |
| IB | AA/! | CB | JB |
| JB | AA/! | DB | KB |
| KB | AA/! | EB | LB |
| LB | AA/! | FB | AA/err |
| CG | AA/err | | DG |
| DG | AA/err1 | | EG |
| EG | AA/err11 | | FG |
| FG | AA/err111 | | GG |
| GG | AA/! | AG | HG |
| HG | AA/! | BG | AA/err |
| GA | AA/! | AA | HA |
| HA | AA/! | BA | IA |
| IA | AA/! | CA | JA |
| JA | AA/! | DA | KA |
| KA | AA/! | EA | LA |
| LA | AA/! | FA | AA/1err |
| BM | AA/err | | CM |
| CM | AA/err1 | | DM |
| DM | AA/err11 | | EM |
| EM | AA/err111 | | FM |
| FM | AA/err1111 | | GM |
| GM | AA/! | AM | AA/err |

auxiliary flag, **low-priority-error** or **high-priority-error,** according to the output buffer currently in use. The comma output shown in the table marks the end of a frame. When a comma is output, the algorithm should do several things before continuing to process input data through the destuffer:

(1) Test the error flag, **low-priority-error** or **low-priority-error,** corresponding to the buffer currently in use. If it is set, clear it. If (and only if) it was already clear, the contents of the buffer should be passed to the data sink (the desired recipient) as a frame of the priority corresponding to the buffer in use.

(2) The contents of the buffer in use should be discarded, making the buffer available for another frame. The output of the two-priority destuffer of Table 2 should be directed to **low-priority-buffer** for successive data. (This may or may not be a change from the buffer previously in use.)

## AN EXAMPLE

Suppose the transmitter received a low-priority frame consisting of the following:*

10011111001111001011111010.

Here I have intentionally included *as part of the data* (at the end of the frame) the flag sequence used by the stuffer of Table 1. This is to emphasize that the data can contain *any* sequence of bits. If a comma is added to this frame and it is then put through the stuffer of Table 1,† the following results, in which brackets have been added to show the bits added by the stuffer:

10011111[00]001111001011111[00]010[011111010].

This differs from the original data in the following ways:

(1) two zeros have been stuffed following the first occurrence of five ones (4th through 8th bits);

(2) two zeros have been stuffed following the second occurrence of five ones (the ones in the flag sequence that was part of the data—this transformation is what makes the inclusion of the flag sequence as part of the data harmless); and

(3) a flag sequence of 011111010 has been added to the end of the frame to represent the comma.

If the stuffed bit sequence as shown (minus brackets) is put through the destuffer of Table 2, the original frame results (with comma appended). The example is more interesting, however, if a high-priority frame is added to the picture. Suppose the following small high-priority frame arrives at the transmitter:

0011010.

Preparing it for transmission by adding a comma, stuffing with Table 1, and prefixing a high-priority flag sequence, the algorithm produces

[1111110]0011010[011111010]

where again the added bits are shown in brackets. Because the frame contained no occurrences of five consecutive ones, no stuff bits were needed and only the delineating flag sequences were added. Now, suppose that the transmission of the low-priority frame is partially completed when this high-priority frame arrives. This causes the (stuffed) sequence of high-priority bits above to be transmitted somewhere in the midst of the low-priority bit stream produced earlier by the stuffing of the low-priority frame. This might look like

10011111000011110010111[1111110001101001111010]11100010011111010

where the high-priority bits are shown in brackets. When this sequence appears at the receiver (without benefit of brackets), it is put through the two-priority destuffer of Table 2. The machine of Table 2 produces the following output:

1001111100111100l!0011010,011111010,

* In a real application most frames would be much longer than this.

† This example was produced with the aid of the test implementation discussed in Appendix B.

which contains the high-priority frame, beginning after the exclamation point and continuing to the first comma. If exclamation points and commas are used to control output buffering, as described earlier, the high-priority frame will be removed to a special **high-priority-buffer** and the remainder, minus punctuation, will end up in the **low-priority-buffer.**

Notice that the high-priority frame at the output of the destuffer of Table 2 is at a slightly earlier point in the low-priority frame than it was when it was inserted into the low-priority data stream at the transmitter. This results because of the information storage in the FSM of Table 2. Any data sequence that happens to be identical with the beginning of a flag sequence is, effectively, stored in the FSM until a bit comes along that shows that a flag sequence is not being received. Consequently, a few bits (three in this example) of the low-priority frame may be stored in the FSM when the high-priority flag sequence arrives. Because those stored bits are not output until the high-priority frame has been completely received, the high-priority frame appears to have moved to an earlier position within the low-priority frame.

## VERIFICATION OF THE FRAME-SYNCHRONIZATION ALGORITHM

### When Only Low-Priority Data Are Involved

The first step in the verification of the two-priority frame-synchronization algorithm is to show that it operates correctly when input consists of low-priority frames only. In this limited context, the handling of the auxiliary variable **Saved** can be simplified. Since **Saved** in Table 2 is initialized to AA, since **Saved** is reset to AA on the zero transition from state BG in Table 2, and since that transition always occurs at the end of a high-priority frame when the destuffer of Table 2 is returning to low-priority operation (as will be seen later), the variable **Saved** will always be set to AA when the destuffer is operating on low-priority data. Therefore, the verification of the correct operation of the stuffer and destuffer together can be carried out with the value of **Saved** assumed to be AA. Construction of an FSM that is the *catenation* of the stuffer and destuffer followed by verification that the input and output of the catenation are identical is sufficient to show that the stuffer and destuffer work correctly together while operating on low-priority data. I constructed the catenation, shown in Table 3, by simulating the operation of the two FSMs with the output of the stuffer of Table 1 applied to the input of the destuffer of Table 2, beginning with both in their starting states. The first two letters of each state designation correspond to the state of the stuffer, and the last two letters of the state designation correspond to the state of the destuffer.

Table 3 — FSM Equivalent to the Catenation of the Stuffer and Destuffer

| State | Input='0' | Input='1' | Input=',' | Stored |
|---|---|---|---|---|
| AAAA | AAAB | BBBA | AAAA/, | |
| AAAB | AAAB/0 | BBBB | AAAA/0, | 0 |
| BBBB | AAAB/01 | CCCB | AAAA/01, | 01 |
| CCCB | AAAB/011 | DDDB | AAAA/011, | 011 |
| DDDB | AAAB/0111 | EEEB | AAAA/0111, | 0111 |
| EEEB | AAAB/01111 | AAAA/011111 | AAAA/01111, | 01111 |
| BBBA | AAAB/1 | CCCA | AAAA/1, | 1 |
| CCCA | AAAB/11 | DDDA | AAAA/11, | 11 |
| DDDA | AAAB/111 | EEEA | AAAA/111, | 111 |
| EEEA | AAAB/1111 | AAAA/11111 | AAAA/1111, | 1111 |

The process of constructing the catenation begins with the state AAAA. State AAAA is equivalent to both the stuffer of Table 1 and the destuffer of Table 2 being in state AA, the starting state. In the beginning, this is the only state the catenation is assumed to be able to reach. When the stuffer receives an input of zero, it moves to (or stays in) state AA and outputs a zero. This zero output goes to the destuffer, where it causes a transition to state AB with no output produced. The entry in Table 3 for the receipt of a zero input from state AAAA should, therefore, show a transition to state AAAB, equivalent to the stuffer being in state AA and the destuffer being in state AB, and should show no output. Once a similar process is used to fill in the other Table 3 entries for transitions from state AAAA, it is apparent that the catenation can reach states AAAB and BBBA as well as its starting state AAAA. Each of these states is added to the first column of the state table of the catenation, and each has entries added for each possible input symbol. This process continues until every state shown in the first column of the table has had entries added on its right for all three input symbols and until each state name appearing as a "next state" appears somewhere in the first column.

On some transitions the stuffer of Table 1 outputs more than one symbol. This has to be handled carefully when the catenation is constructed. Consider, for example, state EEEA in Table 3 with an input of one. The state EEEA is equivalent to the stuffer of Table 1 being in state EE while the destuffer of Table 2 is in state EA. When the stuffer receives an input of one from state EE, it moves to state AA and outputs a one followed by two zeros. The one and the two zeros go to the destuffer, taking it from state EA *through* states FA and AM to state AA and causing it to output five ones along the way. The entry in Table 3 for the receipt of a one input from state EEEA should, therefore, show a transition to state AAAA, equivalent to both the stuffer and the destuffer being in state AA, and should show an output of five ones.

The last column in Table 3 shows the input that is "stored" in the FSM, if any. In other words, this is the cumulative input for which the "appropriate" output has not yet been generated. By "appropriate," I mean the output that should be generated to represent the data if it turns out that the currently-arriving input is not part of a flag sequence.

If the input and the output of the catenation of the stuffer and destuffer can be shown to be identical, then it follows that the destuffer correctly destuffs the output of the stuffer. The value shown in the "Stored" column must be considered in the determination of this equivalence of input and output. For any particular transition in Table 3, the input and output of the FSM are *effectively* equal if the combination of the stored bits followed by the input bit is equal to the output bit(s) followed by the stored bits associated with the *next* state. Figure 1 illustrates this for the zero transition from state EEEB of Table 3. In state EEEB there are five stored bits: 01111. These are the input bits that the FSM has seen but for which no output has yet been generated. The appearance of the zero at the input means the FSM has now seen six bits that it has not acted on: 011110. These six bits are shown to the left of the equal sign in Fig. 1. As it makes the transition from state EEEB to the next state, AAAB, Table 3 shows that the FSM outputs the first five of those six bits, leaving a single zero bit as yet unused. This zero is correctly shown in the "Stored" column of Table 3 for state AAAB.

$$\left[\begin{array}{r} \text{Stored in EEEB}\left\{\begin{array}{c}0\\1\\1\\1\\1\end{array}\right. \\ \text{Input}\left\{\begin{array}{c}0\end{array}\right. \end{array}\right] = \left[\begin{array}{l}\left.\begin{array}{c}0\\1\\1\\1\\1\end{array}\right\}\text{Output}\\ \left.\begin{array}{c}0\end{array}\right\}\text{Stored in AAAB}\end{array}\right]$$

Fig. 1 — Input of Table 3 effectively equals output for zero transition of state EEEA.

If the input and the output of an FSM were effectively equal in this manner for all transitions in the state table of the FSM, the *total accumulated output* of the FSM followed by the bits stored in the machine's final state would always equal the total input the machine had seen. Consequently, the total output of such an FSM would be exactly equal to its total input whenever the machine was in a state having no storage. To summarize this notion: *The output of an FSM must be identical to any input that leaves the FSM in a state with no storage if, for each combination of present state, current input, next state, and output, the present-state stored input catenated with the current input is identical to the output catenated with the next-state stored input.*

The argument that the destuffer of Table 2 correctly destuffs the output of the stuffer of Table 1 when low-priority frames are input now proceeds as follows:

(1) For each combination of present state, current input, next state, and output in the catenated stuffer-destuffer FSM of Table 3, the present-state stored input catenated with the current input is identical to the output catenated with the next-state stored input. Therefore, the total accumulated output (since the FSM began operating) of Table 3, followed by the bits stored in its final state, always equals its total accumulated input.

(2) Since no commas appear in the "Stored" column of Table 3 (implying it cannot store commas), and since the total output of Table 3 followed by the bits stored in its final state always equals its total input, any input sequence ending in a comma appears complete at the output of Table 3, leaving the machine in the only state with no storage: its starting state, AAAA.

(3) All properly formed low-priority frames end with commas. Low-priority frames are, therefore, copied correctly to the output of the FSM of Table 3, and each low-priority frame leaves the machine in its starting state ready to process the next frame.

**When High-Priority Data Are Involved**

To establish the correct operation of the stuffer of Table 1 with the destuffer of Table 2 when high-priority frames are present is more involved. Since each high-priority frame begins with the high-priority flag sequence, and since the entire high-priority frame can be inserted anywhere in the low-priority data, the effects of the high-priority flag sequence on the state of the two-priority destuffer of Table 2 must be determined. Construction of the catenation of the FSMs of Table 1 and Table 2 revealed which states of the destuffer of Table 2 can be reached when it is operating on low-priority data only. These states include not only states whose names appear as second components of state names in Table 3 (states AA, AB, BB, CB, DB, EB, BA, CA, DA, and EA), but they also include those intermediate states of Table 2 used when the destuffer is passed multiple input elements from the stuffer at one time. This happens on the "one" transition from state EE in Table 1 and on all the "comma" transitions of Table 1. The associated transitions of Table 3 are:

(1) the "one" transition from state EEEA, which causes the destuffer to pass through intermediate states FA and AM;

(2) the "one" transition from state EEEB, which causes the destuffer to pass through intermediate states FB and AG; and

(3) the "comma" transitions of Table 3, each of which causes the destuffer of Table 2 to pass through intermediate states AB, BB, CB, DB, EB, FB, AG, and BG.

Therefore, the only states of the destuffer of Table 2 that are used in handling the output of the stuffer in the absence of insertions of the high-priority flag sequence are exactly those states *above* the double line in Table 2.

Simulation of the operation of the destuffer of Table 2 with a high-priority-flag-sequence input, starting from each of those states reachable with low-priority data, produces the result shown in Table 4. The starting states are shown at the left and the sequence of inputs making up the high-priority flag sequence are shown along the top. For each starting state, the sequence of states the destuffer of Table 2 passes through as the flag sequence is input can be read from left to right. The destuffer produces no output on any of these transitions.

Table 4 — Table 2 States Used when a High-Priority Flag Sequence Appears at the Input

| Start | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| AA | BA | CA | DA | EA | FA | GA | AA |
| AB | BB | CB | DB | EB | FB | GB | AA |
| BB | CB | DB | EB | FB | GB | HB | AA |
| CB | DB | EB | FB | GB | HB | IB | AA |
| DB | EB | FB | GB | HB | IB | JB | AA |
| EB | FB | GB | HB | IB | JB | KB | AA |
| FB | GB | HB | IB | JB | KB | LB | AA |
| AG | BG | CG | DG | EG | FG | GG | AA |
| BG | CG | DG | EG | FG | GG | HG | AA |
| BA | CA | DA | EA | FA | GA | HA | AA |
| CA | DA | EA | FA | GA | HA | IA | AA |
| DA | EA | FA | GA | HA | IA | JA | AA |
| EA | FA | GA | HA | IA | JA | KA | AA |
| FA | GA | HA | IA | JA | KA | LA | AA |
| AM | BM | CM | DM | EM | FM | GM | AA |

Looking at the states in the column under the last "one" in the flag sequence in Table 4, and looking at the zero transitions from those states in Table 2, one finds that the last bit of the high-priority flag sequence always sends the destuffer of Table 2 to state AA and sets the auxiliary variable **Saved** to the value shown in the first column of Table 4. Since AA is the correct starting state for the destuffing of a high-priority frame (because it is stuffed exactly like a low-priority frame), the frame following the high-priority flag sequence will be correctly destuffed according to the same arguments used previously to show the correct destuffing of low-priority frames. Since insertions of the high-priority flag sequence are impossible when the destuffer is in the midst of destuffing a high-priority frame, and since the only transition possible that changes **Saved** and can occur in the absence of such insertions is the end-of-frame zero transition from state BG, the value of **Saved** at the end of the high-priority frame is exactly the value stored there at the end of the high-priority flag sequence that initiated the frame: the state the destuffer was in just before the high-priority flag sequence arrived. Therefore, at the end of the high-priority frame the destuffer of Table 2 correctly resumes destuffing the low-priority frame in which the high-priority frame was imbedded.

## SUMMARY

The standard techniques for frame synchronization in bit-oriented link-level protocols are inadequate in situations in which both of the following are true: (1) data can be transmitted only occasionally and in amounts not necessarily corresponding to frames; and (2) data frames are divided into two categories, high priority and low priority, with high priority indicating time-critical information. In these situations, the transmitter data queue should operate as a priority queue; i.e., it should operate in a first-in-first-out fashion except that when high-priority bits enter the queue they immediately move past any low-priority bits already in the queue. Because this can result in a high-priority frame being inserted into the middle of a partially transmitted low-priority frame, the synchronization algorithm in this context needs to do two things in a compatible way: (1) delimit low-priority frames, and (2) mark high-priority frames in such a way that they can be detected anywhere in the low-priority data stream. This report described such an algorithm in detail and showed that its two halves, the transmitter-site stuffer and the receiver-site destuffer, operate correctly together.

## REFERENCES

1. J.W. Conard, "Character-Oriented Data Link Control Protocols," *IEEE Trans. Commun.* **COM-28**(4), 445-454 (Apr. 1980).

2. R.A. Scholtz, "Frame Synchronization Techniques," *IEEE Trans. Commun.* **COM-28**(8), 1204-1213 (Aug. 1980).

3. D.E. Carlson, "Bit-Oriented Data Link Control Procedures," *IEEE Trans. Commun.* **COM-28**(4), 455-467 (Apr. 1980).

4. ANSI Standard X3.66-1979, *Advanced Data Communication Control Procedures*, American National Standards Institute, 1430 Broadway, New York, NY 10018, 1979.

5. B.H. Cantrell, J.O. Coleman, and G.V. Trunk, "Radar Communications," NRL Report 8515, Aug. 26, 1981.

6. F.J. Hill and G.R. Peterson, *Introduction to Switching Theory and Logical Design,* Wiley, New York, 1968.

## Appendix A

## DERIVATION OF THE FRAME-SYNCHRONIZATION ALGORITHM

The derivation of the two-priority frame-synchronization marking and stuffing procedure described in the body of the report (referred to subsequently as simply the *two-priority stuffing* procedure, for brevity) is outlined here as an evolution from the ADCCP stuffing procedure outlined by Carlson (quoted in this report's introduction) to the full two-priority stuffing procedure. First, the ADCCP procedure, described briefly in the Introduction, must be rigorously described.

### The ADCCP Algorithm

Table A1 is a state-table description of a finite-state machine (FSM) that implements the ADCCP stuffing algorithm. The FSM in Table A2 gives the corresponding destuffing algorithm.

Table A1 — FSM Implementation of the ADCCP Stuffer

| State | Input='0' | Input='1' | Input=',' |
|---|---|---|---|
| A | A/0 | B/1 | A/01111110 |
| B | A/0 | C/1 | A/01111110 |
| C | A/0 | D/1 | A/01111110 |
| D | A/0 | E/1 | A/01111110 |
| E | A/0 | A/10 | A/01111110 |

Table A2 — The ADCCP Destuffer

| State | Input='0' | Input='1' |
|---|---|---|
| A | B | I/1 |
| B | B/0 | C |
| C | B/01 | D |
| D | B/011 | E |
| E | B/0111 | F |
| F | B/01111 | G |
| G | A/011111 | H |
| H | A/, | A/err |
| I | B | J/1 |
| J | B | K/1 |
| K | B | L/1 |
| L | B | M/1 |
| M | A | A/err |

In addition to the two data values, the stuffer of Table A1 can accept a comma as input. The comma indicates the end of a frame. The destuffer of Table A2 sees only zeros and ones as input.

Construction of an FSM that is the catenation of the stuffer and destuffer followed by verification that the input and output of the catenation are identical will prove that the ADCCP stuffer and destuffer work correctly together. Each state of the combined FSM corresponds to a state of the stuffer paired with a state of the destuffer. Table A3 shows the combined FSM.

Table A3 — FSM Equivalent to the Catenation of the ADCCP Stuffer and Destuffer

| State | Input='0' | Input='1' | Input=',' | Stored |
|---|---|---|---|---|
| AA | AB | BI/1 | AA/, | |
| AB | AB/0 | BC | AA/0, | 0 |
| BC | AB/01 | CD | AA/01, | 01 |
| CD | AB/011 | DE | AA/011, | 011 |
| DE | AB/0111 | EF | AA/0111, | 0111 |
| EF | AB/01111 | AA/011111 | AA/01111, | 01111 |
| BI | AB | CJ/1 | AA/, | |
| CJ | AB | DK/1 | AA/, | |
| DK | AB | EL/1 | AA/, | |
| EL | AB | AA/1 | AA/, | |

The last column in Table A3 shows the input that is stored in the FSM, if any. In other words, this is the cumulative input for which the appropriate output has not yet been generated. By appropriate I mean the output that should be generated to represent the data if it turns out that the currently-arriving input is not part of a flag sequence.

For each combination of present state, current input, next state, and output in the catenated stuffer-destuffer FSM of Table A3, the present-state stored input catenated with the current input is identical to the output catenated with the next-state stored input. Therefore, the output of Table A3, followed by the bits stored in its final state, always equals its input. Since no commas appear in the "Stored" column of Table A3 (implying it cannot store commas), and since the output of Table A3 followed by the bits stored in its final state always equals its input, any input ending in a comma appears complete at the output of Table A3. Further, since a comma is the last symbol input in a frame, and since each transition on a comma input in Table A3 has state AA as its next state, the FSM is always in state AA at the end of a frame and is ready to begin processing the next frame. This completes the argument that the ADCCP destuffer of Table A2 correctly destuffs the output of the ADCCP stuffer of Table A1.

The ADCCP algorithm just described will be used for end-of-frame marking, with the end of a frame of *either* priority indicated by a comma at the input to the stuffer. Unfortunately, the ADCCP algorithm is not suitable for marking the beginning of the high-priority frame. If the ADCCP flag sequence were inserted into a stuffed data stream at random, it would be possible in certain circumstances for the zero at the beginning of the ADCCP flag sequence to be removed by a destuffer before the flag sequence was recognized. Therefore, a different flag sequence must be adopted to indicate the beginning of a high-priority frame.

## The High-Priority Flag Sequence

To mark the beginning of a high-priority frame, I chose a high-priority flag sequence consisting of six ones followed by a zero. When a high-priority frame is stuffed, this high-priority flag sequence is inserted into the output once at the beginning of the frame and is never used thereafter. When low-priority frames are stuffed, the high-priority flag sequence is never used at all.

Although the fact that the high-priority flag sequence will only be used to mark the *beginning* of a high-priority frame implies that only the low-priority data need to be protected against its coincidental occurrence, it is desirable that the high-priority data be protected as well. This results in the stuffing operations for the high-priority data and the low-priority data being identical, ensuring faster resynchronization if an error in transmission between stuffer and destuffer results in the destuffer losing track of the correct priority of the data.

To see that this is so, consider an example. Suppose the flag sequence at the end of a high-priority frame is not recognized by the destuffer. This will cause the destuffer to process the succeeding low-priority data as if it were high-priority data. If the stuffing processes for high-priority data and low-priority data are identical, the destuffer will recognize the flag sequence at the end of the surrounding low-priority data and believe that it has found the end of the high-priority frame. At this point it will resume processing data, assuming correctly that it is low priority. The net effect of the error was to take the portion of a low-priority frame that followed an embedded high-priority frame and incorrectly catenate it to the high-priority frame.

If different stuffing procedures had been used for high-priority and low-priority data, the nonrecognition of the terminating flag sequence of a high-priority frame would result in a longer resynchronization process. Because the destuffer would process succeeding low-priority data with the wrong destuffing procedure, it would not recognize the flag sequence at the end of the low-priority data. In effect, all the data between the missed flag sequence at the end of the high-priority frame and the terminating flag sequence of the *next* high-priority frame would be incorrectly catenated onto the original high-priority frame. Because high-priority frames would normally be expected to occur less frequently than low-priority frames, this could result in a large loss of data.

The stuffer used to protect a data stream against accidental occurrence of the high-priority flag sequence is identical to the ADCCP stuffer of Table A1. The only difference is in the input that will be fed to the stuffer. The input to this stuffer will come from the output of the stuffer used to mark the end of the frame. This is necessary because a high-priority frame can appear in the midst of a low-priority frame *that is already stuffed,* implying that it can even appear in the middle of the flag sequence used to mark the end of that low-priority frame. Since the ADCCP flag sequence, as well as the data, needs to be protected against coincidental occurrences of the high-priority flag sequence, the high-priority stuffing should apparently be applied at the *output* of the original ADCCP stuffing operation. Because this output contains only ones and zeros, comma inputs need not be considered in the high-priority stuffing operation. Because of the desirability of making the various stuffing operations identical for low- and high-priority data, both low- and high-priority frames will be protected against coincidental occurrences of the high-priority flag sequence by this second stuffing operation.

Because the high-priority flag sequence can appear anywhere in the data stream at the receiver, the role of the destuffer is somewhat different here than it was for the ADCCP destuffer described earlier. To begin with, assume that only an isolated flag sequence will be inserted into the data stream; i.e., it will not be followed by a high-priority data frame. While this restriction will obviously be lifted later, it is useful here to simplify the derivation. Under this assumption the functions of the destuffer are:

(1) to remove the zero stuff bits that were inserted into the data following occurrences of five ones; and

(2) to recognize insertions of the high-priority flag sequence, returning at the completion of the sequence to the state the FSM was in before the arrival of the first bit of the sequence.

The FSM destuffer given in Table A4 performs these functions. It assumes that no comma input is ever used at the corresponding stuffer. An exclamation point is output when a high-priority flag sequence is recognized.

To show that the high-priority destuffer is correct involves several steps. Table A5 shows an FSM that is equivalent to the catenation of the stuffer of Table A1 (without the comma input column) and the high-priority destuffer of Table A4, assuming there are no flag sequence insertions. The output of this combined FSM is always equal to its input catenated with the stored input (by the same argument

Table A4 — The High-Priority Destuffer

| State | Input='0' | Input='1' |
|---|---|---|
| A | A/0 | B |
| B | A/10 | C |
| C | A/110 | D |
| D | A/1110 | E |
| E | A/11110 | F |
| F | A/11111 | G |
| G | A/! | H |
| H | B/! | I |
| I | C/! | J |
| J | D/! | K |
| K | E/! | L |
| L | F/! | L/1err |

Table A5 — FSM Equivalent to the Catenation of the High-Priority Stuffer and Destuffer

| *State* | *Input='0'* | *Input='1'* | *Stored* |
|---|---|---|---|
| AA | AA/0 | BB | |
| BB | AA/10 | CC | 1 |
| CC | AA/110 | DD | 11 |
| DD | AA/1110 | EE | 111 |
| EE | AA/11110 | AA/11111 | 1111 |

used for earlier catenations). A separate check is required to see whether an inserted flag sequence can be recognized without otherwise disturbing the destuffing process. In the absence of the flag sequence in the input, the destuffer of Table A4 uses only states A through F. Proper handling of the flag sequence requires that if the machine begins in one of these six states, a flag sequence input will leave the machine back in the same state after it outputs only an exclamation point. The FSM of Table A4 meets this condition.

## The Two-Priority Algorithm

At this point the necessary raw material has been developed for the needed marking and stuffing algorithm. It remains only to put the pieces together. The two-priority algorithm will first be summarized as a combination of the FSMs just developed. The algorithm will then be condensed into a more compact and efficient equivalent. The two-priority stuffing algorithm using the ADCCP stuffer, summarized in Fig. A1, consists of the following:

(1) Take the data frame (of either priority), follow it with a comma, and put it through the ADCCP stuffer of Table A1. This marks the end of the frame with the ADCCP flag sequence and stuffs the data stream to prevent accidental occurrence of that flag sequence.

(2) Take the output of the ADCCP stuffer (*including* the flag sequence at the end of the frame) and put it through the ADCCP stuffer again. This second pass stuffs the data to prevent accidental occurrence of the high-priority flag sequence. Because the input to this second pass consists only of ones and zeros, the output of this second stuffing pass will not contain any flag sequences. (The flag sequence from the first stuffing pass will have a zero stuffed into it during this second pass.)

(3) If (and only if) this is a high-priority frame, precede it with the high-priority flag sequence.

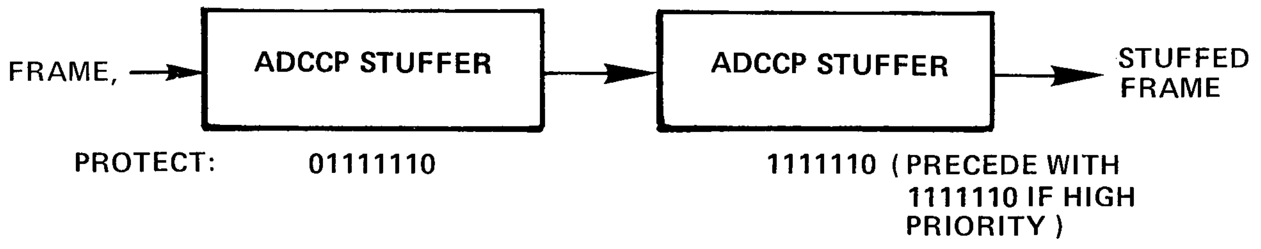

Fig. A1 — Two-priority stuffing with two stuffers

The two-priority destuffing algorithm is more involved. It involves the use of two output streams, low priority and high priority, and the use of two copies each of the high-priority destuffer of Table A4 and the ADCCP destuffer of Table A2. Figure A2 shows the relationship between the four destuffers and should be referred to during the following discussion. Destuffing takes place as follows:

(1) Begin by putting the received data stream into a high-priority destuffer whose output goes to an ADCCP destuffer whose output is low priority. These two destuffers are shown at the top of Fig. A2.

(2) The exclamation point output by the high-priority destuffer on receipt of a high-priority flag sequence should not be sent to the ADCCP destuffer, but it should be used instead as a signal to switch the input stream to the other copy of the high-priority destuffer. This second copy sends its output to an ADCCP destuffer whose output is high priority.

(3) When this second ADCCP destuffer outputs a comma, the input stream should be returned to the original high-priority destuffer, which then takes up destuffing from the state it was last in. The comma is *not* removed from the output stream.

It is not difficult to see intuitively that the stuffing and destuffing procedures just described are compatible. Proof is not given here, however, because the final form of the algorithm is proven in the body of the report.

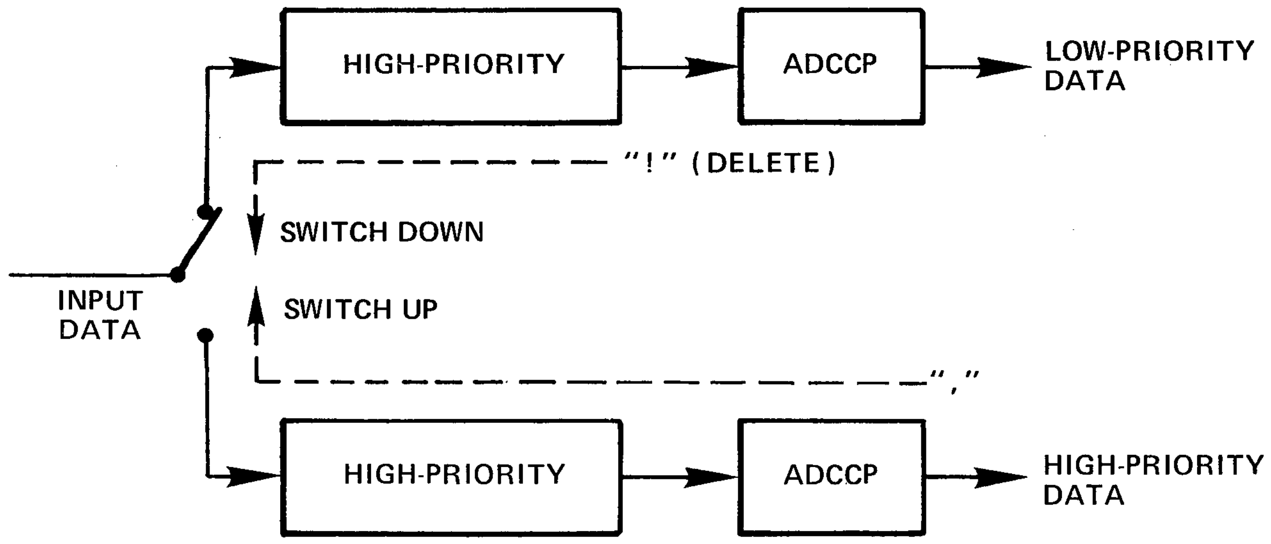

Fig. A2 — Two-priority destuffing with four destuffers

## Simplifying the Algorithm

The algorithm at this point involves six identifiable FSMs, two at the transmitter and four at the receiver. The remaining steps in the derivation reduce the number to two, one each for the transmitter and the receiver. To combine the two stuffers into one equivalent "double stuffer," catenate the FSM of Table A1 with itself. Table 1 in the body of the report shows this catenation. Inspection of the table shows that this FSM is, in effect, an ordinary stuffer that uses a flag sequence of a zero, five ones, a zero, a one, and a zero. To prevent accidental occurrence in the data of both this sequence and the high-priority flag sequence, it inserts a double zero following the occurrence anywhere in the data of five ones.

The first step in the simplification of the two-priority destuffer is the catenation of the high-priority destuffer of Table A4 with the ADCCP destuffer of Table A2 into a single FSM. Table A6 gives the resulting "double destuffer." The two-priority destuffing algorithm can be constructed from two of these double destuffers, as shown in Fig. A3. (It may be instructive to compare Figs. A2 and A3.) In the catenation all outputs of "!" or "err" from the high-priority destuffer (the first FSM in the catenation) were assumed to pass straight through to the output of the ADCCP destuffer without affecting its state. Wherever a transition of the catenated FSM of Table A6 shows an output containing an "err" indication, the associated next state may or may not appear in the table, as the next states for these transitions are the result of arbitrary choices made in the design of the destuffers of Tables A2 and A4, and they will be changed later anyway.

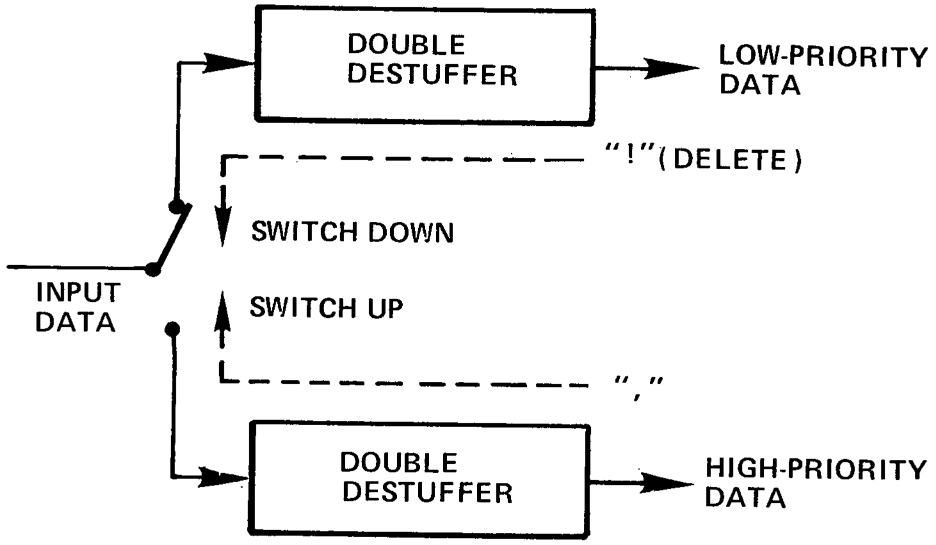

Fig. A3 — Two-priority destuffing with two destuffers

Because there are constraints on what can appear at the input of this machine, it is reasonable to hope that some of the states can be eliminated as unreachable with legitimate inputs. To determine exactly what states are reachable, it is appropriate to go ahead and form the catenation of the double stuffer of Table 1 with the double destuffer of Table A6. Performing the catenation will show exactly which states of the double destuffer of Table A6 can be reached in the absence of insertions of the high-priority flag sequence. The result of the catenation is given in Table 3 in the body of the report.

The first two letters of each state designation correspond to the state of the double stuffer and the last two letters of the state designation correspond to the state of the double destuffer. In addition to the states implied by Table 3, the double destuffer of Table A6 passes through some intermediate states

Table A6 — The Double Destuffer

| State | Input='0' | Input='1' |
|---|---|---|
| AA | AB | BA |
| AB | AB/0 | BB |
| BB | AB/01 | CB |
| CB | AB/011 | DB |
| DB | AB/0111 | EB |
| EB | AB/01111 | FB |
| FB | AG | GB |
| AG | AA/011111 | BG |
| BG | AA/, | CG |
| BA | AB/1 | CA |
| CA | AB/11 | DA |
| DA | AB/111 | EA |
| EA | AB/1111 | FA |
| FA | AM/11111 | GA |
| AM | AA | BM |
| GB | AB/! | HB |
| HB | BB/! | IB |
| IB | CB/! | JB |
| JB | DB/! | KB |
| KB | EB/! | LB |
| LB | FB/! | LC/err |
| CG | AB/err | DG |
| DG | AB/err1 | EG |
| EG | AB/err11 | FG |
| FG | AK/err111 | GG |
| GG | AG/! | HG |
| HG | BG/! | IG |
| IG | CG/! | JG |
| JG | DG/! | KG |
| KG | EG/! | LG |
| LG | FG/! | LH/err |
| GA | AA/! | HA |
| HA | BA/! | IA |
| IA | CA/! | JA |
| JA | DA/! | KA |
| KA | EA/! | LA |
| LA | FA/! | LI/1err |
| BM | AB/err | CM |
| CM | AB/err1 | DM |
| DM | AB/err11 | EM |
| EM | AB/err111 | FM |
| FM | AL/err1111 | GM |
| GM | AM/! | HM |
| HM | BM/! | IM |
| IM | CM/! | JM |
| JM | DM/! | KM |
| KM | EM/! | LM |
| LM | FM/! | LA/errerr |

whenever it receives multiple input elements from the double stuffer at one time. This happens on the "one" transition from state EE in Table 1 and on all the "comma" transitions of Table 1. The associated transitions of Table 3 are:

(1) the "one" transition from state EEEA, which causes the double destuffer to pass through intermediate states FA and AM;

(2) the "one" transition from state EEEB, which causes the double destuffer to pass through intermediate states FB and AG; and

(3) the "comma" transitions of Table 3, each of which causes the double destuffer of Table A6 to pass through intermediate states AB, BB, CB, DB, EB, FB, AG, and BG.

Therefore, the states of the double destuffer of Table A6 that are necessary and sufficient to handle the output of the double stuffer in the absence of insertions of the high-priority flag sequence are exactly those states *above* the double line in Table A6. To complete the determination of the necessary states of Table A6 under the implied input constraints, it is now necessary to simulate the operation of the FSM of Table A6 with a high-priority flag sequence as input, starting from each state that can be reached in the absence of insertions of the flag sequence (those states above the double line). Table A7 shows the result of this simulation. The starting states are shown at the left, and the sequence of inputs making up the high-priority flag sequence is shown along the top. For each starting state, the sequence of states the double destuffer of Table A6 passes through as the flag sequence is input can be read from left to right, along with any output the double destuffer produces. The states that could not be reached in the absence of insertions of the flag sequence are shown in italics. Comparison of Table A7 with the double destuffer FSM of Table A6 shows that states IG, JG, KG, LG, HM, IM, JM, KM, and LM of Table A6 are superfluous and can be eliminated, since they cannot be reached with inputs produced by the associated stuffing scheme.

Table A7 — Table A6 States Used When a High-Priority Flag Sequence Appears at the Input

| Start | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| AA | BA | CA | DA | EA | FA | *GA* | AA/! |
| AB | BB | CB | DB | EB | FB | *GB* | AB/! |
| BB | CB | DB | EB | FB | *GB* | *HB* | BB/! |
| CB | DB | EB | FB | *GB* | *HB* | *IB* | CB/! |
| DB | EB | FB | *GB* | *HB* | *IB* | *JB* | DB/! |
| EB | FB | *GB* | *HB* | *IB* | *JB* | *KB* | EB/! |
| FB | *GB* | *HB* | *IB* | *JB* | *KB* | *LB* | FB/! |
| AG | BG | *CG* | *DG* | *EG* | *FG* | *GG* | AG/! |
| BG | *CG* | *DG* | *EG* | *FG* | *GG* | *HG* | BG/! |
| BA | CA | DA | EA | FA | *GA* | *HA* | BA/! |
| CA | DA | EA | FA | *GA* | *HA* | *IA* | CA/! |
| DA | EA | FA | *GA* | *HA* | *IA* | *JA* | DA/! |
| EA | FA | *GA* | *HA* | *IA* | *JA* | *KA* | EA/! |
| FA | *GA* | *HA* | *IA* | *JA* | *KA* | *LA* | FA/! |
| AM | *BM* | *CM* | *DM* | *EM* | *FM* | *GM* | AM/! |

There is some question about the appropriate choices of next states to be associated with "err" indications in the output. The next state indications in Table A6 are those arising directly from the catenation of the high-priority destuffer and the ADCCP destuffer. Since these two component destuffers had next state indications that had been chosen completely arbitrarily for those transitions with an "err" in the output, there is no reason to keep the resulting transitions in the double destuffer of Table A6. There are several possible approaches to the selection of these next states:

(1) Identify, for each transition involving an error indication, the possible input (transmission) errors that could have caused the problem, and identify the one that was the most likely. The FSM is then designed to go the state it would be in if the error so identified had never occurred.

(2) The assumption could arbitrarily be made that the error occurred in the bit just received. This implies that the next states should be the same for both inputs from any present state in which either input causes an error indication in the output.

(3) Always return on an "err" transition to the starting state AA. This, effectively, assumes that there is no way that any one assumption about what went wrong and led to the error is better than any other.

Without meaning to imply that it is the preferred approach, I have chosen the last for simplicity. This choice, of course, could easily be changed.

The remaining step in the derivation of the two-priority destuffer is the combining of the two destuffers of Fig. A3 into one. Notice that a single output stream from the two-priority destuffer is enough if it is specified that the arrival of an exclamation point at the output always redirects the output to the high-priority data stream and the arrival of a comma always redirects the output to the low-priority data stream. Figure A4 illustrates this. As in Figs. A2 and A3, the exclamation point itself is deleted from the output while the comma is not. The comma goes to whichever output stream is in effect *before* the switch is thrown. Under these conditions a single FSM can be found to duplicate the function of the two destuffers of Figure A3 if each state of the combined FSM corresponds to a pair consisting of one state from each destuffer. This would be similar to the replacement of the lower destuffer of Fig. A3 with 15 separate destuffers, one for each of the 15 next states associated with an exclamation point output in Table A6. Since the upper destuffer would have all 39 reachable states of Table A6, and since each of the 15 lower destuffers would have the 15 states of Table A6 that can be reached without an insertion of the high-priority flag sequence, the combined FSM would have 264 states. All this can be avoided, however, by the use of a single 39-state machine to perform the functions of all 16 of the component machines just enumerated. An auxiliary variable stores the extra state information required. The two-priority destuffer using an auxiliary variable named **Saved** is shown in Table 2 in the body of the report.

The rationale for the handling of the variable **Saved** is as follows: When an exclamation point is output from the upper destuffer in Fig. A3, input is switched to the lower destuffer, which always begins in its starting state, AA. Therefore, all transitions in Table A6 with an exclamation point output have been given a next state of AA in Table 2, with the next state shown in Table A6 saved in the auxiliary variable **Saved.** When the lower FSM of Fig. A3 outputs a comma, the upper FSM of Fig. A3 must begin operating where it left off. This is accomplished in the two-priority destuffer of Table 2 by the machine's transitioning to the state saved in the variable **Saved** whenever a comma is output. In addition, the variable **Saved** is set to state AA (the next state associated with the zero transition of state BG in Table A6) whenever a comma is output, insuring the appropriate behavior when the comma at the end of a low-priority frame is output (i.e., any comma is output from the upper FSM of Fig. A3).*

* The variable **Saved** serves a function similar to the storage of the return address in a subroutine, function, or procedure call in a computer program.

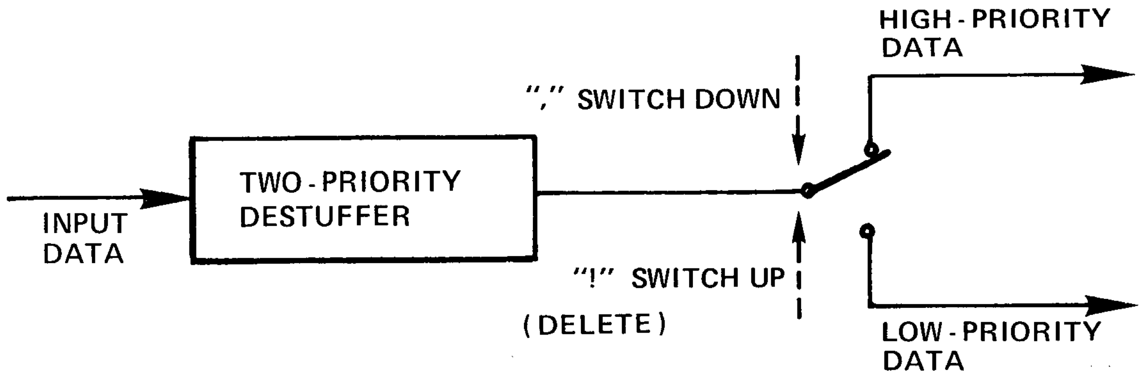

Fig. A4 — Output control with two-priority destuffer

## Appendix B

## A TEST IMPLEMENTATION

The algorithm described in this report has not yet been implemented in a way suited to operational use. A "quick-and-dirty" implementation was constructed, however, so that the algorithm could be tested. Because the approach taken in the test implementation was somewhat unconventional, I decided to present it here as an alternative to the use of Table 1 and Table 2 as discussed in the body of the report.

The first step in the implementation of the stuffer and destuffer was the conversion of their state tables to formal grammars. The grammars were then used as specifications to a compiler-generator called yacc [B1,B2]. A compiler-generator is a computer program that accepts a specification as input and produces as output another computer program or a portion of a program. It is most commonly used to create the parsing portion of a compiler, hence the name. Here I have used yacc to create two programs implementing the equivalents of Table 1 and Table 2.

Figure B1 gives the grammar used as a yacc specification of the double stuffer.* Figure B1 is interpreted as follows: Each line contains a grammar rule followed by an associated action enclosed in braces. The first two lines are exceptions to this pattern and will be discussed separately later. The two-letter designations appearing in the rules are the equivalent of the states of Table 1. That portion of a rule to the left of the colon is the *left side* of the rule. For those lines that start with the indented vertical bar, the left side is implied to be the same as the left side of the last line that did *not* begin with the bar. In Fig. B1, for example, there are twelve rules with left sides of AA. That portion of the rule to the right of the colon or vertical bar is the *right side* of the rule. While the left side always consists (for this type of grammar) of the name of a state (usually termed a *nonterminal symbol* in a grammar), the right side may consist of a sequence of state names and possible input symbols (usually termed *terminal symbols),* with the input symbols enclosed in single quotes to distinguish them clearly from the state names. Each rule specifies one legitimate way for the machine to proceed in a particular context. The left side gives the *state* that is in some sense "equivalent" to the *condition* specified by the right side. Because it is equivalent, the machine will transition to the state given on the left side when the condition specified by the right side is met. The right side can be interpreted if we consider each symbol as an input string. In particular, the name of a state appearing on the right side of a rule is equivalent to any input string that could leave the machine in that state. For example, the line in Fig. B1 whose left side is CC can be interpreted as: proceed to state CC once input that leaves the machine in state BB is followed by an input of 1. This is entirely equivalent to the "Input=1" transition out of state BB in Table 1.

The first line in Fig. B1 makes it legal for the machine to transition to a "STOP" state from any of the other states on encountering the end of the input data; end-of-data is thus never considered an error. The rule on the second line contains no right side, making it legal to proceed to state AA without any input having appeared at all. This is equivalent to specifying the state AA as the starting state of the machine.

All the rules in Fig. B1 but the first two are followed by associated actions enclosed in braces. Each action is just a statement in the C programming language [B3,B4] that should be executed when the associated rule changes the state of the machine. For this test implementation, each action is a

* The yacc-specification grammar given in Fig. B1 was created directly from a computer-stored version of Table 1 by the use of global editing operations.

```
STOP     : AA | BB | CC | DD | EE ;   /* allow machine to accept any input */
AA       :   /* empty */              /* corresponds to FSM starting state */
         | AA ','      {    printf( "011111010" ) ;      }
         | AA '0'      {    printf( "0" ) ;              }
         | BB ','      {    printf( "011111010" ) ;      }
         | BB '0'      {    printf( "0" ) ;              }
         | CC ','      {    printf( "011111010" ) ;      }
         | CC '0'      {    printf( "0" ) ;              }
         | DD ','      {    printf( "011111010" ) ;      }
         | DD '0'      {    printf( "0" ) ;              }
         | EE ','      {    printf( "011111010" ) ;      }
         | EE '0'      {    printf( "0" ) ;              }
         | EE '1'      {    printf( "100" ) ;            } ;
BB       : AA '1'      {    printf( "1" ) ;              } ;
CC       : BB '1'      {    printf( "1" ) ;              } ;
DD       : CC '1'      {    printf( "1" ) ;              } ;
EE       : DD '1'      {    printf( "1" ) ;              } ;
```

Fig. B1 — Yacc specification for the double stuffer of Table 1

statement that prints the output associated with the corresponding transition of Table 1. It is not difficult to verify that each line (after the first two) in Fig. B1 has a corresponding transition in Table 1 and vice versa.

Figure B2 gives the grammar used as a yacc specification for the two-priority destuffer of Table 2. Its interpretation is the same as described earlier for Fig. B1 with two exceptions. First, there is an additional subtlety related to the **Saved** variable of Table 2. In this implementation there is no explicit **Saved** variable. The reason is that the program generated by yacc uses a stack to store state information, and the information that is stored in **Saved** in Table 2 is stored here on the stack.

An example should make clearer the parallel between Table 2 and Fig. B2 where **Saved** is involved. Look at the first row following the double line in Table 2. This shows that, on receiving an input of zero from state GB, the two-priority destuffer should output an exclamation point, set **Saved** to AB, and then transition to state AA. The machine should then run exclusively in the states above the double line (because of input constraints) until a zero input is received while in state BG. This zero input forces the machine to output a comma and transition to the state whose name is stored in **Saved,** which in this case is state AB.

The equivalent of this process is represented in Fig. B2 by the last rule whose left side is AB. This rule, and several others similar to it, takes up two lines of Fig. B2. (Notice that the second of these lines does not begin with the bar.) Its right side contains two actions rather than one. The first two items on the right side represent an input of a single zero following an input that could result in the machine arriving at state GB. This much is parallel to the two-priority destuffer of Table 2 being in state GB with a zero at the input. To continue with the right side of the Fig. B2 rule, ignore the first action temporarily; the BG on the second line of the rule represents an input string that leaves the machine in state BG. This implicitly includes *the entire input* processed by the two-priority destuffer of Table 2 beginning immediately after it left state GB (in our example) and ending as it subsequently arrived in state BG. This is because the only way the machine can reach state BG is by first being at state AA and then processing input, changing state as many times as necessary, until state BG is reached. Since Fig. B2 has a rule that says that state AA is equivalent to a null (empty) input, the transition from state GB to state AA can proceed regardless of what input symbol follows the first zero on the right side of the rule. If the entire condition specified by the right side of the rule does occur, the machine represented by Fig. B2 interpolates the two actions shown on the right side at the points corresponding to their placement, then it transitions to the "equivalent" state AB. The machine of

```
STOP    : AA | AB | AG | AM
        | BA | BB | BG | BM
        | CA | CB | CG | CM
        | DA | DB | DG | DM
        | EA | EB | EG | EM
        | FA | FB | FG | FM
        | GA | GB | GG | GM
        | HA | HB | HG | IA
        | IB | JA | JB | KA
        | KB | LA | LB ;        /* allow machine to accept any input */
AA      :   /* empty */         /* corresponds to FSM starting state */
        | AG '0'                   { printf( "011111" ) ;  }
        | AM '0'
        | FM '0'                   { printf( "err1111" ) ; }
        | GA '0'                   { printf( "!" ) ;       }
                BG '0'             { printf( "," ) ;       }
        | GM '1'                   { printf( "err" ) ;     }
        | HG '1'                   { printf( "err" ) ;     }
        | LA '1'                   { printf( "1err" ) ;    }
        | BM '0'                   { printf( "err" ) ;     }
        | CG '0'                   { printf( "err" ) ;     }
        | CM '0'                   { printf( "err1" ) ;    }
        | DG '0'                   { printf( "err1" ) ;    }
        | DM '0'                   { printf( "err11" ) ;   }
        | EG '0'                   { printf( "err11" ) ;   }
        | EM '0'                   { printf( "err111" ) ;  }
        | FG '0'                   { printf( "err111" ) ;  }
        | LB '1'                   { printf( "err" ) ;     } ;
AB      : AA '0'
        | AB '0'                   { printf( "0" ) ;       }
        | BA '0'                   { printf( "1" ) ;       }
        | BB '0'                   { printf( "01" ) ;      }
        | CA '0'                   { printf( "11" ) ;      }
        | CB '0'                   { printf( "011" ) ;     }
        | DA '0'                   { printf( "111" ) ;     }
        | DB '0'                   { printf( "0111" ) ;    }
        | EA '0'                   { printf( "1111" ) ;    }
        | EB '0'                   { printf( "01111" ) ;   }
        | GB '0'                   { printf( "!" ) ;       }
                BG '0'             { printf( "," ) ;       } ;
AG      : FB '0'
        | GG '0'                   { printf( "!" ) ;       }
                BG '0'             { printf( "," ) ;       } ;
AM      : FA '0'                   { printf( "11111" ) ;   }
        | GM '0'                   { printf( "!" ) ;       }
                BG '0'             { printf( "," ) ;       } ;
BA      : AA '1'
        | HA '0'                   { printf( "!" ) ;       } .
                BG '0'             { printf( "," ) ;       } ;
BB      : AB '1'
        | HB '0'                   { printf( "!" ) ;       }
                BG '0'             { printf( "," ) ;       } ;
```

Fig. B2 — Yacc specification for the two-priority destuffer of Table 2

```
BG      : AG '1'
        | HG '0'                {  printf( "!" ) ;       }
                BG '0'          {  printf( "," ) ;       } ;
BM      : AM '1' ;
CA      : BA '1'
        | IA '0'                {  printf( "!" ) ;       }
                BG '0'          {  printf( "," ) ;       } ;
CB      : BB '1'
        | IB '0'                {  printf( "!" ) ;       }
                BG '0'          {  printf( "," ) ;       } ;
CG      : BG '1' ;
CM      : BM '1' ;
DA      : CA '1'
        | JA '0'                {  printf( "!" ) ;       }
                BG '0'          {  printf( "," ) ;       } ;
DB      : CB '1'
        | JB '0'                {  printf( "!" ) ;       }
                BG '0'          {  printf( "," ) ;       } ;
DG      : CG '1' ;
DM      : CM '1' ;
EA      : DA '1'
        | KA '0'                {  printf( "!" ) ;       }
                BG '0'          {  printf( "," ) ;       } ;
EB      : DB '1'
        | KB '0'                {  printf( "!" ) ;       }
                BG '0'          {  printf( "," ) ;       } ;
EG      : DG '1' ;
EM      : DM '1' ;
FA      : EA '1'
        | LA '0'                {  printf( "!" ) ;       }
                BG '0'          {  printf( "," ) ;       } ;
FB      : EB '1'
        | LB '0'                {  printf( "!" ) ;       }
                BG '0'          {  printf( "," ) ;       } ;
FG      : EG '1' ;
FM      : EM '1' ;
GA      : FA '1' ;
GB      : FB '1' ;
GG      : FG '1' ;
GM      : FM '1' ;
HA      : GA '1' ;
HB      : GB '1' ;
HG      : GG '1' ;
IA      : HA '1' ;
IB      : HB '1' ;
JA      : IA '1' ;
JB      : IB '1' ;
KA      : JA '1' ;
KB      : JB '1' ;
LA      : KA '1' ;
LB      : KB '1' ;
AA      : BG '0'                {  printf( "," ) ;       } ;
```

Fig. B2 (Continued) — Yacc specification for the two-priority destuffer of Table 2

Fig. B2, therefore, operates in a manner parallel to the two-priority destuffer of Table 2 for all those transitions of Table 2 that set the **Saved** variable below the double line in the table. This is because all those transitions operate analagously to the example just discussed.

There is one Table 2 transition involving **Saved** that was not covered by analogy in the above discussion. If the two-priority destuffer of Table 2 is processing low-priority data when it arrives at state BG, the variable **Saved** will contain the value AA. The zero transition corresponding to this situation is represented in Fig. B2 by the last rule, which shows that state AA is equivalent to an input that can leave the two-priority destuffer in state BG, followed by an input of zero. The second area in which Fig. B2 is interpreted differently from Fig. B1 involves this rule. The right side of this rule is identical to the last portion of several other rules (the exceptional rules discussed above). When the condition implied by the right side of the last rule appears, it will sometimes be true that the condition implied by the right side of one of these other rules is also true. In these cases the ambiguity is resolved by the application of whichever applicable rule occurs *first* in the grammar specified in Fig. B2. This results in the test implementation represented by Fig. B2 behaving exactly according to the two-priority destuffer specification of Table 2.

## REFERENCES

B1. S.C. Johnson, "Yacc — Yet Another Compiler-Compiler," Computer Science Technical Report 32, Bell Laboratories, Murray Hill, New Jersey, July 1975.

B2. A.V. Aho and J.D. Ullman, "An Automatic Parser Generator," in *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts, 1977, pp. 229-241.

B3. D.M. Ritchie, S.C. Johnson, M.E. Lesk, and B.W. Kernighan, "UNIX Time-Sharing System: The C Programming Language," *Bell Syst. Tech. J.* **57**(6), 1991-2019 (1978).

B4. B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.